COSMOS: A System-Level Modelling and Simulation Framework for Coprocessor-Coupled Reconfigurable Systems

Wu, Kehuai; Madsen, Jan

Published in:
International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, 2007. IC-SAMOS 2007.

Link to article, DOI:
10.1109/ICSAMOS.2007.4285743

Publication date:
2007

Document Version
Publisher's PDF, also known as Version of record

COSMOS: A System-Level Modelling and Simulation Framework for Coprocessor-Coupled Reconfigurable Systems

Kehuai Wu, Jan Madsen
Dept. of Informatics and Mathematical Modelling, Technical Univ. of Denmark
Email: {kw,jan}@imm.dtu.dk

***Abstract*— Dynamically reconfigurable systems demand complicated run-time management. Due to resource constraints and reconfiguration latencies, efficient reconfiguration strategies that can reduce the overhead cost of dynamic reconfiguration need to be studied. In this paper, we i) propose a reconfigurable task model which extends the classical real-time task model to support the additional states and latencies needed to capture dynamically reconfigurable behavior, ii) propose a coprocessor-coupled reconfigurable architecture which has hardware run-time support for task execution, task reallocation and resource management, and iii) present a SystemC based framework to model and simulate coprocessor-coupled reconfigurable systems. We illustrate how COSMOS may be used to capture the dynamic behavior of such systems and emphasize the need for capturing the system aspects of such systems in order to deal with future design challenges of dynamically reconfigurable systems.**

I. Introduction

Future embedded systems will be based on platforms which allow the system to be extended and incrementally updated while running in the field. This will not only extend the life time of the system, but also allow the system to adapt to the physical environment as well as performing self-repair and hence increasing the reliability and robustness of the system. In order to facilitate this, the platforms need to be dynamically reconfigurable architectures. Although these platforms will be based on multiprocessor system-on-chips (MPSoC) and network-on-chip (NoC) architectures, the dynamic behavior of the hardware pose new challenges to tools and methodologies in order to ensure both efficient platform design and run-time platform usage.

Reconfigurable architectures fully exploit the tradeoff between the chip area and hardware reusability. Instead of implementing the digital IP core with fixed logic, a programmable digital device is used. By switching the application running on the programmable device at run-time, the architecture should have the flexibility of software and the efficiency of the hardware, thus enables us to close the gap between the software and the hardware.

The biggest challenge in reconfigurable system design is to improve the rate of reconfiguration at run-time by reducing the reconfiguration overhead. The reconfiguration overhead comes from multiple sources, and without proper management, the flexibility of the reconfiguration can not justify the overhead cost. Many new technologies and designs for minimizing the reconfiguration overhead have been proposed. Logic granularity [4], [5], host coupling [3], resource management [7], [6] etc. have been studied in various contexts. These technologies substantially increase the practicality of the reconfigurable systems, but also often lead to highly complicated system behavior. There exists several highly efficient architectures, but many of them have significant drawbacks in terms of programmability, flexibility, scalability or utilization rate.

Even though low-level technologies have drawn a lot of attention, the study on system-level behavior and compilation is still in its infancy. As high level design decisions made early in the design process can have a high impact on the performance of reconfigurable systems, the evaluation of applications executing on a reconfigurable system in the early development stages, is a new challenge which needs to be addressed.

For datapath-coupled architectures [11], [4], reconfigurable unit (RU) is frequently designed as a special instruction-set functional unit or extended to a large-scale VLIW processor, thus the application can be efficiently evaluated with instruction-level simulation. However, coprocessor-coupled architectures, which are usually large-scale, need advanced run-time resource management and carefully designed architectures. Hence, to improving the system efficiency, we need to be able to model and analyze such architectures and the applications running on them.

In this paper we present COSMOS, a framework to model and simulate coprocessor-coupled reconfigurable systems. We propose a novel real-time task model which captures the additional characteristics to correctly handle dynamically reconfigurable systems. We also propose a general model of coprocessor-coupled reconfigurable systems. The task and architecture models are based on an existing MPSoC simulation model, ARTS [1], which has been extended to facilitate run-time resource management strategies. To the best of our knowledge, this is the first attempt to create a system-level modelling framework for dynamically reconfigurable systems, which takes into account the reconfigurable architecture, the application running on the architecture and the run-time task and resource management.

The rest of the paper is organized as following. Section II gives an overview of the new technologies employed in reconfigurable system design. Section III proposes the recon-

1-4244-1058-4/07/$25.00 ©2007 IEEE

figurable system task model which captures the real-time application execution characteristics. Section IV propose a real-time system model of the coprocessor-coupled architecture with focus on the run-time resource management and task execution efficiency. Section V discusses how the task and architecture models have been implemented in SystemC, and a demonstrative simulation result is presented in section VI. Section VII discusses our model's future work and Section VIII concludes our study.

II. Background

During a reconfiguration, reconfigurable architectures suffers from latencies due to context switching (configuration and intermediate data) of an RU. The severity of this latency is determined by several physical factors, e.g. the scale of the RU, the logic granularity, the configuration memory bandwidth, the rate of reconfiguration or the buffering technics of reconfiguration memory fetching. In the following we will give an overview of the related research areas that can reduce such latencies, and discuss how they affect system behavior.

One research trend assume that the applications, or a collection of tasks, share the RU in time, as shown in Figure 2A. [8] proposed a multi-context FPGA that can significantly reduce the reconfiguration time, but the extra cost of chip area is hardly justifiable by the performance gain. A solution that can substantially reduce the area overhead is to increase the logic granularity of the RU to medium- or coarse-grained, as shown in Figure 1. Even if these higher-granularity architectures do not offer highly optimal solutions to applications that heavily exploit bit-level data manipulations, the concept of multi-context is proved feasible. But still, the number of contexts being cached on the RU is usually limited, and optimal utilization of the limited context resource at run-time is a difficult challenge for a multi-tasking system [9].

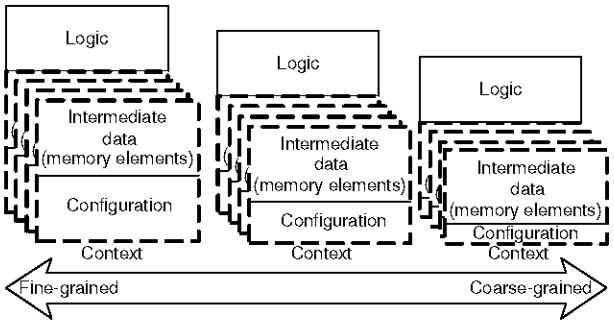

Fig. 1. The impact of logic granularity on the chip area of reconfigurable architectures.

Another type of reconfigurable architecture assume that the RU is shared in space [10], [7], as shown in Figure 2B. The RU is partially reconfigurable and large-scaled, thus several tasks can be run on the same RU with no conflicts among each other. Besides reconfiguration latency, this class of architectures leads to complicated inter-task communication and resource management. Since a task can be allocated on an RU at any free location during run-time, data traffic between tasks go through non-deterministic paths, maybe requiring dynamic routing. For a large programmable array, the complexity of performing the task placement and data routing at run-time can be very hard to handle. Also, it is clear that the fragmentation is a common issue for this kind of design, thus task (context) reallocation and rerouting is consistently required for defragmentation. In summary, the behavior and efficiency of such system can be very unpredictable, and understanding the system behavior in the early development stage is crucial.

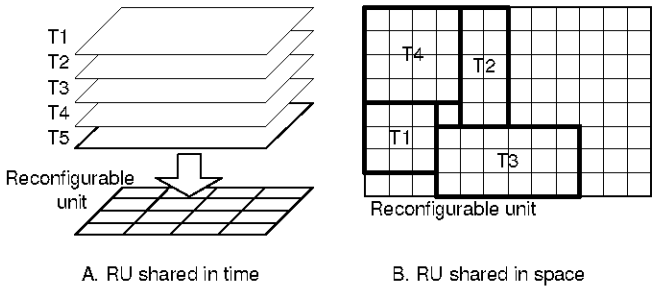

Fig. 2. Reconfigurable unit design

A third type of RU is a hybrid of the two former families. This type of architecture is one of the main focus of our work, and it will be introduced in section IV. This architecture is viewed as an array of networked multi-context RUs. Such system also requires efficient dynamic resource management, but the routing problem is greatly simplified compared to space-shared architectures.

In general, we are facing the increasing complexity of the reconfigurable system's spatial and temporal behavior. New technologies that improve the system's efficiency also complicates the architecture, and the value of the tradeoff between performance and design complexity is not easily assessable. A system-level simulator is much needed for evaluating the performance of dynamically reconfigurable systems. Such a simulator should give the designer the opportunity to tune various design parameters and to study the consequences on system performance.

To build a system-level simulator, we need a thorough understanding on how to model the tasks which comprise the application. A task running on the reconfigurable architecture has a different execution behavior than the classical real-time task, thus the classical model does not fit our purpose. For the simulator design, we need i) a general and generic model of the RU which can represent various types of coprocessor-coupled RU designs, ii) the dynamic resource management issue of the RU should be addressed, and iii) the simulation should be parameterizable so that the consequences of changes in the physical design can be captured within the model.

The ARTS modelling Framework captures real-time behavior of heterogeneous multiprocessor systems, where each processor may run its own operating system. In our work, we adopt the underlying message-passing based mechanism of the ARTS model and some of its RTOS functionality. We extend it

further to support the modelling of dynamically reconfigurable systems. In particular, our model, unlike ARTS, supports task reconfiguration and reallocation during rune-time, i.e. during simulation.

III. Task Model

In the ARTS framework, an application is modelled as a set of task graphs, and each task is modelled as a finite state machine (FSM), as shown in Figure 3. The state transitions of a task are driven by the operating system control messages. Whether a task should run, or be preempted, depends on the resource allocation, scheduling and task dependency. But for reconfigurable system, such an FSM is not sufficient to capture the task execution scenario.

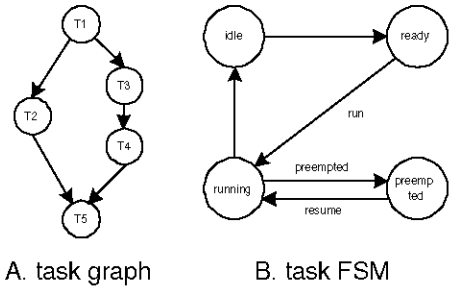

Fig. 3. ARTS task model

Firstly, to initialize the task execution, the initial configuration needs to be loaded from the configuration memory to the RU. Depending on the RU's granularity, size and memory interfacing, the timing cost for fetching the whole configuration can be a big overhead. To explicitly express this task execution phase, a new state **init_config** has been added to the task model, as shown in Figure 4.

Secondly, the preemption is not simply a process of task giving up an RU for other tasks, but is also a process of hardware context switching. Differ from the software context switching, which mostly involves backing up special-purpose registers and bookkeeping the operating system management entries, hardware context switching needs to back up the configuration and all the intermediate data stored in all the memory elements. The timing cost of the preemption can be extremely high if the context is stored in the external memory, or as low as a single clock cycle if the RU has multi-context support. Architecture designers would experiment on various combinations of different context storage design in order to find an optimal strategy, thus the reconfiguration latency may vary. In our model, we added two delay states, **reconfig_preempt** and **reconfig_run**, to represent the timing cost of the preemption.

Finally, the effect of process of task migrating among multiple RUs need to be modelled. As shown in Figure 4, we add the reallocation state **realloc** at three places and marked it with dashed circles, in order to emphasize that this single state has multiple entry points and exit points. This is modelled so

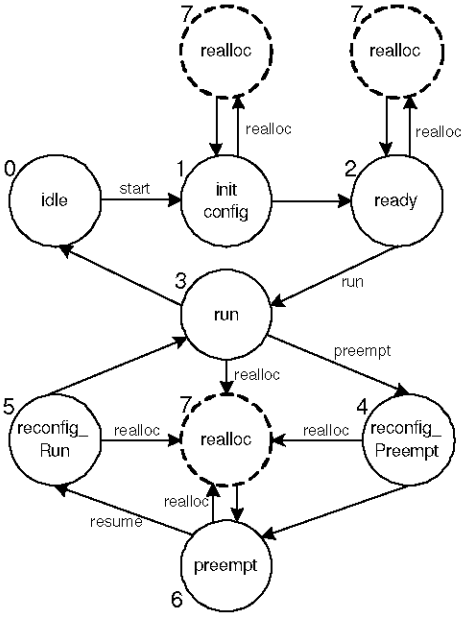

Fig. 4. Real-time reconfigurable task model

because reallocation can happen anytime after a task leaves the idle state, and at different point of time, the reallocation has different effect on task execution. If the reallocation is started before a task is run for the first time, the task needs to be initialized on a different RU. In this case, the (partial) context of the reallocated task is moved to another RU, then it resumes the previous state for either continue initializing or waiting to get permission to run. If the task has been run before, then the allocation must be ended with the task going to the preempt state. The reason for such a setup is because the task doesn't know if it can continue executing after the reallocation, since the resource status of the reallocated RU is unknown. It is safe for a task to preempt itself and request resource management unit for permission to continue execution.

IV. Coprocessor Coupled Architecture Model

As shown in Figure 2, the architecture design is heading to two directions. Besides the aforementioned resource management issue, the time-shared architectures also suffer from scalability issue, since parallelizing a task to use the full RU gets harder when RU's size increases. Similarly, the space-shared architecture's defragmentation gets harder when the RU upscales, and the rerouting becomes impossible to handle at run time. Unless the RU is partitioned and modularized, the space-shared architecture has too many practical issue to realize.

To solve the problem of both types of the reconfigurable architecture, we propose a hybrid architecture model. As shown in Figure 5, our coprocessor consists of an array of homogeneous multi-context RUs connected with on-chip networks (NoC). Similar to the time-shared design, the computation resource of our architecture is still the context of

the RUs. By statically partitioning the applications into tasks, each of which is small enough to fit into one RU's context, or one resource, we can explore the application's parallelism at different levels and efficiently utilize the potential of the coprocessor. To ease the dynamic resource management, we assume that tasks never share one RU in space, even if several tasks can fit into one RU at the same time. This guarantees that each task can be reallocated without interfering the execution of the other tasks.

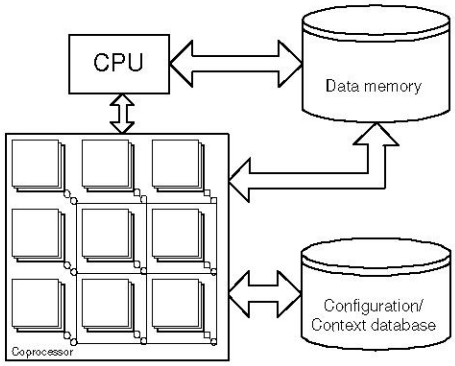

Fig. 5. Hybrid coprocessor-coupled reconfigurable system

As an architecture design, our architecture has several advantages compared to many previous designs. Our coprocessor is upscaled by increasing the number of RUs, thus has more flexibility to efficiently support tasks of various complexity. With the support of the NoC, rerouting problem can be solved on the fly when the tasks are communicating. Since the coprocessor is modularized, defragmentation is not a crucial issue as in previous space-shared design. When combined with our resource management strategy, which will be discussed later, our co-processor is highly scalable.

As a model, our model can easily be used for both time-shared and space-shared architecture PSE. To model the time-shared architectures, by assuming the number of RUs to be one, our model imitates a multi-context architecture. As to model a space-shared architecture, by assuming all the RUs to be single-context, our model can be viewed as a modularized space-shared RU. By employing a NoC and assuming that any task can be allocated to a randomly selected RU, the dynamic placement and routing issues of space-shared architecture becomes a much easier issue to address. However, the homogeneity of RUs adds an extra compile/synthesis resource constraint.

The resource management is still a problem for our architecture, since the run-time system needs to manage tasks in both space and time. For a small-scaled coprocessor, the CPU/operating system can be used to manage the resource. But if the system reaches certain scale, it is foreseeable that taking a snapshot of the whole coprocessor's resource distribution, evaluating it and allocating/reallocating task by using the CPU can be a performance bottleneck. Here we introduce our alternative to address this issue.

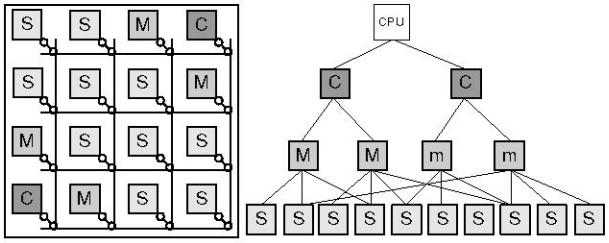

Fig. 6. Hierarchical organization of reconfigurable units

As shown in Figure 6, some nodes in the coprocessor are selected as Coordinator nodes (C-nodes) or Master nodes (M-nodes), and the rest are Slave nodes (S-nodes). By structuring the whole design into hierarchies, the resource management is distributed into different roles each type of the nodes play.

C-nodes are the resource management nodes. Each time an application is started by CPU, all C-nodes send message to the lower hierarchy for resource check. Then M-nodes collect the weighted resource distribution status from S-nodes and pass it to the C-nodes. Then the C-nodes, all of which run the same decision-making protocol, select a resource-optimal M-nodes to initialize and synchronize the application's execution.

M-nodes are the task execution management nodes. After the C-nodes assign an application to an M-node, the M-node reallocates the currently running tasks to free up some resources if the new application has a higher priority. Then the M-node initializes the new application's tasks to free resources, and start its execution. During execution, depending on the task dependencies and priorities, the M-node can reallocate the tasks or preempt the task execution. C-nodes and M-nodes forms a cluster. M-nodes is only controlled by the C-nodes in its cluster, thus any message received from other C-node will be ignored.

S-nodes are the computation units. When a task is allocated to an S-node, the task can be blocked or selected for execution, depending on its priority or deadline. The node keeps track of how many resources is currently in use and how many is still available. The multi-context S-nodes is not bounded to one specific M-nodes. As long as it gives optimal results, contexts on a S-node can be shared among all M-nodes.

The tasks of a certain application are distributed on the S-nodes near one M-node selected by the C-nodes. The higher priority an application has, the more effort the M-nodes will put into to cluster its tasks, in order to lower the communication cost. Lower priority applications' clusters can be disrupted by the M-nodes when a new application with higher priority is started. Careful placement of clusters can help achieve overall system optimality, thus is crucial in our approach.

Our hierarchical design represents our general resource management strategy, but we don't enforce a physical bounding between the function of a resource/task management unit and a RU, except for the S-nodes. For instance, when the coprocessor is small, the function of the C-node and M-nodes

can be realized by the operating system running on CPU, or be combined into one physical RU. It gives us the freedom to model the architecture on various scales. In our experiment, priority is based on the overall communication demand of an application, but we don't constrain how task priority is defined or what allocation strategy is used. Different designer may have different preference on specific parts of the system, and we leave them open for experimentation.

Even though it is not the focus of our work, the latency of off-coprocessor data communication can be easily modelled by our framework. Data IO ports connected to the main memory can be modelled as S-nodes with no context limitation, and off-coprocessor data communication can be modelled as special tasks that can only be allocated on the S-nodes that imitates the coprocessor's IO ports. Given a set of coordinates to the ports, which is preferably on the boundary of the coprocessor, the off-coprocessor communication latency can change when the task reallocation occurs, depending on the distance between the IO port and the tasks that need access to the main memory.

We will not go into detail of the NoC model design in our work, since it has been addressed by the ARTS model described in previous work[2]. In this paper, we simply assume that if two communicating tasks are k hops away on the coprocessor, the communication latency is kT, where T is the single-hop base communication latency between those tasks decided through static analysis. The overall communication latency of an application is greatly affected by the allocation strategy, which is one of the most interesting issue to be addressed by our model.

When a task is reallocated, the context of a task is transferred from one RU to another. This process results in a burst of data transfer on the NoC in a short period of time. Compared to the context transfer, inter-task communication happens much more frequently than reallocation, and data is often delivered in smaller packets. These two types of data transmission have very different requirements on the NoC design, thus we separate them into two NoCs. The reallocation NoC is assumed to be able to establish preset paths that can guarantee to finish the reallocation in a short period of time, thus the physical distance between the context transfer's source and destination should not play a significant role in the overall reallocation latency. In our model, we assume that any reallocation takes a constant period of time, and several reallocation can take place concurrently without blocking each other. Physically, the configuration data communication could share the inter-task communication NoC, but to demonstrate the concept, we choose not to do so at the moment.

One final note about our architecture is, compare to previous works, our approach relies more on the static analysis. We assume that all the RUs are homogeneous, which implies that when an application is partitioned into a task graph, each task has to be able to optimally utilize the computation power of the RU. It is a challenge to perform DSE with several design constraints imposed by the RU design, especially if the high-level synthesis is used.

V. System-C simulation model

The general structure of our System-C model is shown in Figure 7. Various types of modules are organized as mentioned in Figure 6 and connected with communication links defined in the System-C master-slave communication library. The links in solid line are used to convey resource allocation control messages, while the links in dashed lines are for task execution control message passing. The critical design issue of our model is to support task allocation, task execution and task reallocation.

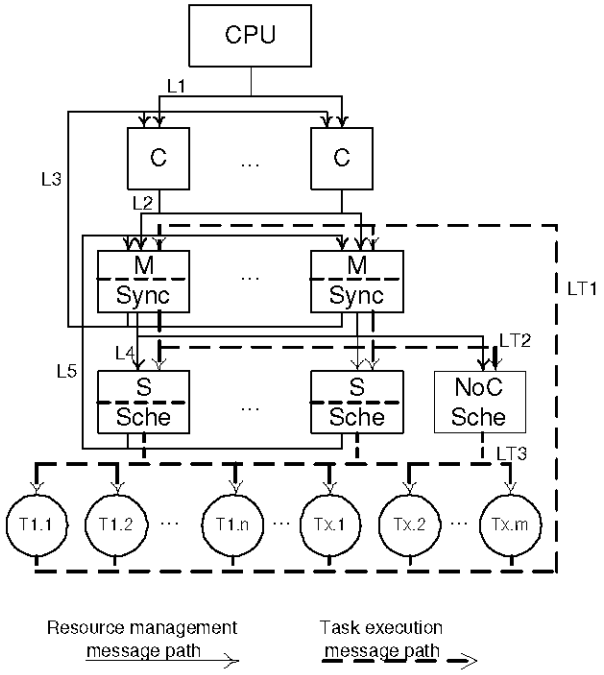

Fig. 7. COSMOS model structure

A. Task allocation

When CPU requests to execute an application, it sends out a message that includes the header description of the application to all the C-nodes through link L1. The information contained in the header are the application's **allocation priority**, **default distribution requirement**, **distribution matrix** and the **application size**. Applications with higher **allocation priority** can force low **allocation priority** tasks to be reallocated and give up resources. The **default distribution requirement** is an integer that specifies the number of S-nodes needed for optimal allocation of an application. For instance, the task graph in Figure 3A will be optimally executed if it is allocated on 2 S-nodes, due to its task-level parallelism. The **distribution matrix** specifies how the tasks are divided into groups, each of which should be allocated on the same S-node. For example, the task graph in Figure 3A can have a **distribution matrix= [**[T1, T3, T4],[T2, T5]]. This indicates that, in order to optimally utilize the task level parallelism and minimize the communication cost, M-node should attempt to allocate task 1, 3 and 4 on one S-node, and allocate task 2 and

5 on the other one. **Application size** simply stands for how many tasks the application has been partitioned into.

Upon receiving the message that requests C-nodes to start up an application, each C-node will further send request through link L2 to the M-nodes in their clusters to exam the resource distribution. M-nodes send the request further down to S-nodes through link L4, and each S-node reports how many free context it has to M-nodes through link L5.

At this point, each M-node has an updated resource distribution map of the whole coprocessor, and needs to evaluate if the M-node itself is resource-abundant. Application is optimally allocated if its tasks are nested into cluster, thus having clustered free resources around an M-node ease the allocation for this M-node. Depending on the resource distribution around a specific M-node, resources are weighed for this M-node. Another factor that influence the allocation is the reallocation potential of each M-node. If there are many high-priority applications nested around and being controlled by an M-node, reallocation will be difficult to perform for this M-node. Thus, we sum up the priorities of all the running tasks being controlled by each M-node, and use it to downgrade the overall resource count. To summarize, each M-node weighs its resource distribution map and sums up all the weighed resource to get an overall weighed resource count, then the number is divided by the priority sum of all the running tasks.

After the M-nodes calculate their final resource count, the number is sent to the C-nodes through link L3. C-nodes then decides which M-node has the highest amount of resource available for the application. Together with the **application priority** and the **distribution matrix**, the decision is then passed through link L2 to the selected M-node for setting up the task execution.

The selected M-node first attempts to reallocate some running tasks, whose priorities are relatively lower than that of the new application, to free up some S-nodes till there is enough free clustered context to allocate the newly-started application. Then the new application's tasks are allocated to the free S-nodes with the guidance of **distribution matrix**. If the **distribution matrix** can not be strictly followed due to the resource availability, spanning to several more S-nodes is allowed. After each task is allocated onto an S-node, the M-node sends a "start" message to all these tasks through link L4, the corresponding S-node and link LT3 to signal the task execution. And finally, some bookkeeping is done in M-nodes and S-nodes.

B. Task execution control

The task execution in COSMOS is essentially the same as in ARTS multiprocessor model. Task execution is controlled through the synchronizer in the M-node and the scheduler in the S-node, as shown in Figure 7. In COSMOS, we adapt to the well-understood direct synchronization (DS) protocol and the earliest-deadline-first (EDF) scheduling for initial experimentation.

Task interacts with the run-time system in the similar way as ARTS tasks model. When a task is ready for execution, it sends a "ready" message to synchronizer through link LT1. When the synchronizer and scheduler permit the task execution to start, the task receives a "run" message through link LT3 and starts executing. When the task is in the "run" state, it can be preempted by the scheduler at any point of time. Similarly, when the task is in the "preempted" state, scheduler can issue "resume" message to let the task continue executing.

The synchronizer acts as a task dependency filter. Its purpose is to block the execution of those tasks that have unsolved dependencies to the preceding tasks. When a task is initialized and has requested for initial execution through link LT1, the synchronizer will immediately block the task if there are unsolved data dependencies. Every time a task finishes its execution, the synchronizer check if any blocked task's dependency is completely solved and ready for execution. The M-node is selected to perform the synchronization since it has the control of the whole application.

When a task is released by the synchronizer, a message is sent from the synchronizer to the scheduler through link LT2. The EDF scheduler then decide if the task should start the execution on the S-node immediately or be blocked until the currently running task is finished, depending on which task's deadline is arriving earlier. If the task released by the synchronizer has a tighter deadline compared to the currently running task, the currently running task is preempted and blocked in a task list. Once the running task has finished its execution, the blocked task that has the earliest deadline is selected for execution.

C. Task reallocation

As mentioned before, task reallocation can occur anytime between the time the task starts initialization and the end of execution. The reallocation basically involves putting the task into the reallocation state for a period of time and updating the task model's information about onto which S-node it is reallocated. The reallocation of a task goes through several different scenarios when the reallocation is initiated at different point of time.

The first possibility is the case that a task is reallocated during initialization. In this case, the task hasn't been blocked by either the synchronizer or scheduler, and the task goes back to initialization right after the reallocation is finished.

The second possible case is the situation that a task is reallocated when it is in the ready state. In this case, the task might be blocked by either the scheduler or the synchronizer. When the task goes into the reallocate state, the synchronizer or the scheduler that blocks the reallocated task need to clean up the record of blocking. When the task finishes the reallocation, the task sends the "ready" message again to the synchronizer to get processed again and goes into the "ready" state again. It is worth noting that, if a task is blocked by the synchronizer when the reallocation start, it is not necessarily true that the same task is still blocked by the synchronizer when the reallocation is finished, since the task dependency can be solved during the reallocation process.

The last possibility is the case that a task has been partially executed before reallocation. The task can only be blocked by the scheduler, or not be blocked at all. If the task is blocked by a scheduler, the scheduler also needs to clean up the record of task being blocked. After the task is reallocated, the task goes into the "preempted" state and send a "ready" message to synchronizer, which will directly pass the message to the scheduler where the task is reallocated since the task dependency has been solved before.

D. NoC model and communication tasks

The ARTS framework explicitly models the communication latency between tasks if the tasks are allocated on different processing elements. As shown in Figure 8, communication between tasks are treated in two different ways. A local communication inside of a processor, e.g. the dependency between T1 and T3, is assumed to have no timing cost, and the dependency is implicitly solved by the local synchronizer. But the communication between tasks allocated on different processors are transformed into communication tasks with explicit execution time, e.g. as task c1_2. A NoC scheduler is used as shown in Figure 8 and 7 to handle the communication latency and NoC scheduling strategy. The communication latencies are decided before simulation time.

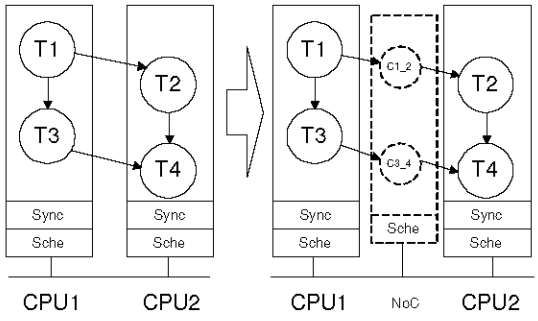

Fig. 8. ARTS communication task

In COSMOS model, since the tasks can be reallocated at simulation time, any task dependency can become a communication task. Furthermore, the communicating source and destination is not fixed on the RU array, thus the physical system and the model should have a varying communication latency. In our model, we assume all the task dependencies to be a communication task, and each communication task has a base latency. Each time a task is reallocated, the communication task that is linked to the reallocated task update its communication source or destination's coordinates, depending on how the task is linked to the communication task. If a communication task's source and destination are allocated on the same S-Node, the communication will be finished in one simulation clock cycle, which is negligible. If the source and destination of a communication task are not allocated on the same S-node, the communication latency is the product of the base communication task latency multiplied by the number of hops between the source and the destination s-nodes.

VI. Simulation results

To demonstrate the function of the model, we set up the architecture and application as shown in Figure 9. The architecture is a 3x3 RU array with one C-node, one M-node and seven S-nodes, each of which supports dual-context. Application 1, 2, and 3, whose task graphs are shown in Figure 9C, start their execution at t=T1, T2 and T3, respectively, as shown in Figure 9A. The application 1 is assigned a slightly earlier deadline compared to the other 2 applications for demonstrative purpose. We assume all the communication tasks to have a single-hop latency of two clock cycles, and the NoC scheduler can only handle one communication message at a time. The latencies for task initial configuration and task reallocation are assumed to be 5 cycles. The latencies of task staying in **reconfig_preempt** state and **reconfig_run** state are assumed to be 3 cycles. All the numbers presented here, including the size of architecture and various timing figures, are only for demonstration purpose and only serves the purpose of helping readers to understand the function of the model. COSMOS is a flexible model, and there is no constraints on how these number can be decided.

An optimal system's reallocation strategy should minimize the occurrence of task reallocation while keeping the overall communication overhead small. But for our experiment, in order to demonstrate the scenario of task reallocation with a simple setting, we select a reallocation strategy that is far from optimal. We define the M-node to be the only cluster center for all three applications. When each application is initialized, its task will be allocated as close to the M-nodes as possible, resulting in the lower priority task to be reallocated on the S-nodes farther away from the M-node. This is achieved by weighing the resource with the distance between the S-node and the M-node during resource evaluation, selecting the most resource-optimal S-node for allocation and selecting the second-most resource-optimal S-node for reallocation.

At t=14, CPU requests to start application 1. The M-node first check the application's distribution matrix for allocation guidance. According to the application 1's distribution matrix, which suggests that task T0_0 and T0_1 should be allocated on the same RU, the M-node initialize both tasks on S-node(0,2). Task T0_2 and T0_3 are both allocated on S-node(1,1) for the same reason. After the tasks finishes the initialization and get ready for execution, only task T0_0 goes into the "run" state, since it's the only task without unsolved dependencies. All the other tasks are blocked by the synchronizer for the time being.

At t=22 the application 2 is initialized. Since the application 2 has the same priority as application 1, it does not cause any task reallocation. At t=30, application 3 starts its initialization. Since this application has a higher allocation priority, previously allocated tasks have to be reallocated to more remote S-nodes. As shown in Figure 9B, task T0_0, T0_1 and T0_2 are replaced by task T2_0, T2_1 and T2_2, respectively. From the waveform, we can see that the reallocated tasks enter and exit "realloc" state at the same time. Since task T0_0 is running when being reallocated, after it finishes the reallocation, it

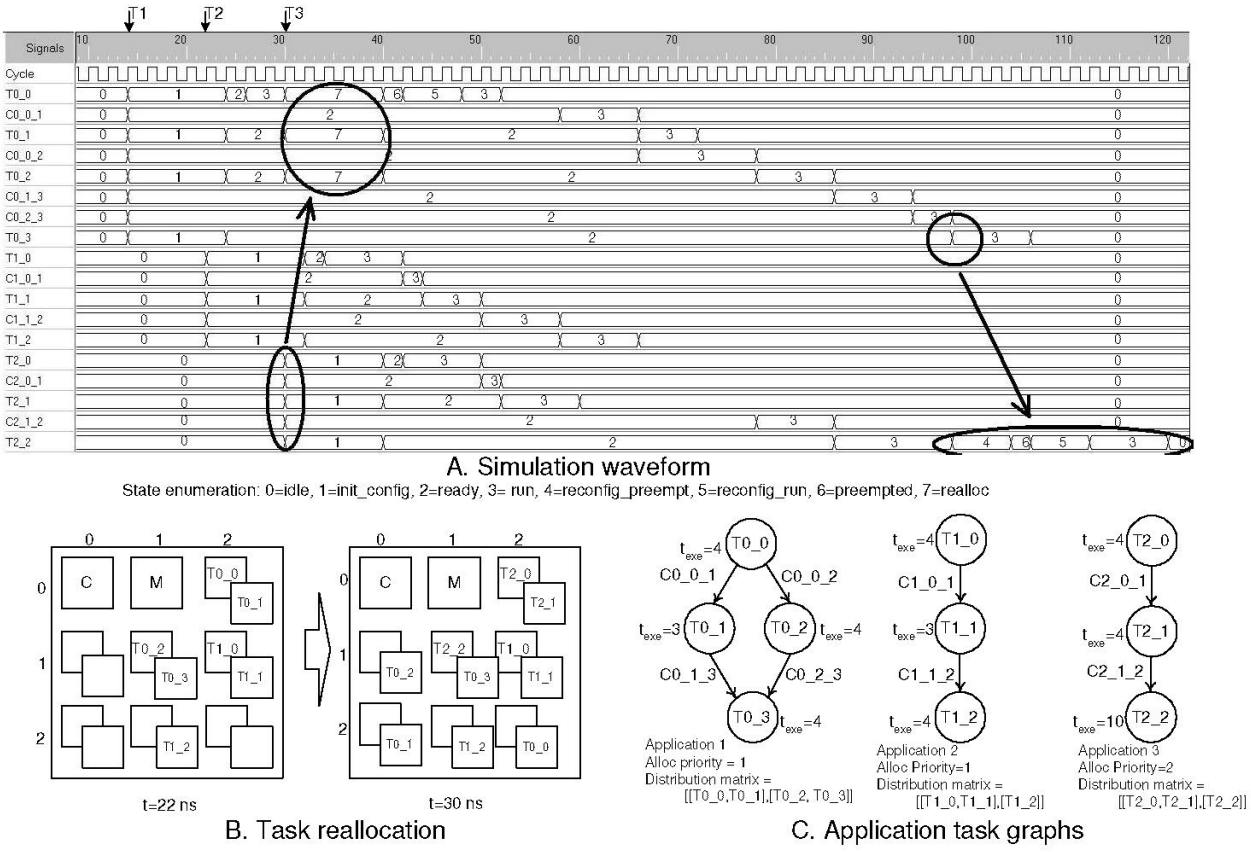

Fig. 9. Demonstrative simulation

goes into "preempted" state and wait for synchronizer and scheduler to start it again, as shown in Figure 4. The other two reallocated tasks go back to "ready" state and wait for their dependencies to be solved.

After the reallocation, communication task c0_0_1 and c0_2_3 become non-local, and the communication task c0_0_2's latency is increased by one hop. The communication task c2_0_1, which is made local by the distribution matrix and reallocation, cost only one clock cycle to finish.

At t=98, task T2_2 goes through a few state changes, which is caused by task T0_3. As shown in Figure 9B, these two tasks are allocated on the same S-node. At t=86, task T2_2's dependency is solved, and the synchronizer starts its execution. When the simulation time reaches 98, task T0_3's dependency is also solved, and the scheduler decides that T0_3 should start its execution since it has an earlier deadline. Task T2_2 goes through a long preempt phase and return to the "run" state after task T0_3 finishes its execution.

VII. Future Work

Our current work concentrates on the real-time behavior of the task being executed on the coprocessor, but the timing characteristic of the management strategy has not been thoroughly addressed. Take reallocation for an example, the decision of which task should be reallocated to which S-node is an important decision. If the M-node can thoroughly analyze the current resource distribution status, the reallocation strategy will improve the result. However, the decision is made by M-node at run-time, thus the latency on making the decision becomes an overhead to the task execution as well. There are trade-offs between reallocation optimality and the reallocation latency, and such trade-off is not well-understood at the moment. In our model, we currently assume that all the resource management algorithms are executed in no time, which can be too optimistic for a large system. In the future, some improvement can be made on this aspect of our model.

One other major issue is the methodology design. In our work, we proposed to execute tasks on a homogeneous RU array, which requires area/resource-constrained partition and synthesis tool to support the architecture. Not only the task-level parallelism need to be analyzed when designing the task graph, but in order to optimally utilize the RU of the given size and resource, we also have to investigate the loop-level parallelism. Also, task partition has direct impact on the inter-task communication latency, which is a crucial factor on task execution efficiency. In the future we will study some benchmarks and investigate how challenging it is to partition an application into equal-sized hardware tasks in order to optimally utilize our architectures.

From the application allocation/execution scenario, we can identify plenty of issues to be addressed in the future. The

strategies of allocation, scheduling and reallocation et.c. are open for further study, and the architecture design issues in terms of NoC and C/M-node design are still not thoroughly addressed. We are currently looking into programmable soft-processor for the C/M-node implementation, which gives the possibilities of reconfiguring a physical S-node into C-node or M-node, to achieve self-reparation.

VIII. Conclusion

We presented a general real-time execution model for tasks that are executed on the reconfigurable architecture. Then we proposed a general system-level real-time simulation model of the coprocessor-coupled reconfigurable architectures. Our simulation framework is highly scalable, and can be extended to support various run-time management algorithms and communication strategies. Through our simulation, we demonstrated how our simulator can be used for studying the system-level design, and pinpointed what architecture design issues can impact the application execution performance.

Reconfigurable system usually are highly complicated to analyze at an early development stage, and the need of simulation tool support is crucial in order to understand the interplay between the architecture, the application and the run-time management system. Our future work will focus on the run-time management system development and the design space/platform space exploration of the reconfigurable systems with our framework.

References

[1] J. Madsen, K. Virk and M. J. Gonzalez *A SystemC-based Abstract Real-Time Operating System Model for Multiprocessor System-on-Chips* Multiprocessor System-on-Chips pp 283-311 Morgan Kaufmann 2004

[2] J. Madsen, S. Mahadevan, K. Virk *Network-Centric System-Level Model for Multiprocessor System-on-Chip Simulation* Interconnect-Centric Design for Advanced SoC and NoC, Springer, 2004

[3] S. H. Katherine Compton. *Reconfigurable computing: A survey of systems and software.* ACM computing surveys, pages 171.210, 2002.

[4] B. Mei, S. Vernalde, D. Verkest, H. D. Man and R. Lauwereins *ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix,* International Conference on Field Programmable Technology, Hong Kong, December 2002, pages 166-173.

[5] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings, 1999. *A reconfigurable arithmetic array for multimedia applications.* ACM/SIGDA International Symposium on FPGAs, 135143

[6] A. Sudarsanam, M. Srinivasan and S. Panchanathan *Resource Estimation and Task Scheduling for Multithreaded Reconfigurable Architectures* Proceedings of the Tenth International Conference on Parallel and Distributed Systems (ICPADS04)

[7] C. Steiger, H. Walder, and M. Platzner, *Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks* IEEE TRANSACTIONS ON COMPUTERS, VOL. 53, NO. 11, NOVEMBER 2004

[8] S. Trimberger, D. Varberry, A. Johnson and J. Wong, *A Time-Multiplexed FPGA* The 5.th Annual IEEE Symposium on FPGAs for Custom Computing Machines. 1997 proceedings, page 22-28

[9] H, Liu and D. F. Wong *A Graph Theoretic Optimal Algorithm for Schedule Compression in Time-Multiplexed FPGA Paretitioning* IEEE/ACM International Conference on Computer-Aided Design, 1999. page 400-405

[10] www.xilinx.com *Two Flows for Partial Reconfiguration: Module Based or Difference Based XAPP290* V1.2 September 9, 2004

[11] A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo, R. Guerrieri *A VLIW processor with reconfigurable instruction set for embedded applications* Solid-State Circuits, IEEE Journal of Volume 38, Issue 11, Nov. 2003 Page(s):1876 - 1886